Combining Parallel and Sequential Workloads on a Network of Workstations

Remzi Arpaci, Amin Vahdat, Thomas Anderson, and David Patterson

October 25, 1994

Abstract

This paper examines the plausibility of using a network of workstations (NOW) for a mixture of parallel and sequential jobs. Through trace-driven simulation, our study identifies a number of results that should be of interest to NOW system designers. First, it is not sufficient to use workstation resources to provide a supercomputer only by night. Next, parallel programs can cause a significant number of lengthy delays to interactive users. Finally, simple scheduling techniques can identify available workstations and minimize user delays while providing parallel program performance comparable to a dedicated massively parallel processor. If these scheduling policies are employed, parallel programmers and interactive users can peacefully coexist on a NOW.

1 Introduction

Rapid improvement in workstation performance has resulted in widespread interest in using networks of workstations (NOWs) for parallel processing [Gelernter 1985, Kronenberg et al. 1986, Carriero & Gelernter 1989, Sunderam 1990, Blumrich et al. 1994]. Because of the economies of scale, building parallel computers from mass-produced hardware and software components can be more cost effective than building the system from scratch. Though massively parallel processor (MPP) interconnects today have better performance than local area networks (LANs), emerging standards such as ATM [Biagioni et al. 1993] and the repackaging of MPP interconnects as LANs [Cohen et al. 1993] makes NOWs more attractive, even for communication intensive parallel programs.

To date, NOWs have been used only for batch-style programming on a dedicated cluster of workstations, although idle machines represent a substantial untapped resource. As a consequence, parallel programmers and interactive users often use logically similar but physically distinct computing platforms, leaving both systems idle much of the time. Two reasons account for this distinction in computing platforms. First, scheduling sequential and parallel jobs on a single platform is difficult because parallel programs must be co-scheduled [Ousterhout 1982, Gupta et al. 1991, Feitelson & Rudolph 1992] to achieve acceptable communication performance. Co-scheduling ensures that a parallel job simultaneously runs on each of its assigned processors, allowing the job's communication requests to be serviced without a context switch. Second, the interactive response times that have made personal computing popular may be compromised if computing resources are shared with demanding applications. As an example, many users of SRC's distributed load balancing system, *dp*, were found periodically tapping their keyboards to keep their workstation from being harvested by the system.

This paper examines the plausibility of using a NOW for a mixture of parallel and sequential jobs. The goal of our research is to build a NOW that runs parallel programs with the same

performance as a dedicated MPP and runs sequential programs with the same performance as a dedicated uniprocessor. To ensure that interactive response time is preserved, parallel jobs are run only on idle workstations. To simplify the implementation of co-scheduling, parallel jobs are scheduled only when there are enough idle idle workstations to run the entire job. Thus, a 32-node parallel job will not run unless there are at least 32 idle workstations in the cluster. As a result of these two basic assumptions, active sequential users are not slowed by parallel programs, and parallel programs can be co-scheduled to achieve acceptable performance.

Currently, NOWs do not simultaneously execute sequential and parallel workloads because of a number of logistical difficulties. First, the metric used to identify idle workstations must be accurate, otherwise both sequential and parallel jobs are significantly slowed. Next, users must wait for the eviction of parallel jobs and the restoration of their workstation state upon return to their machine: these delays can be quite lengthy and bothersome. Finally, a lack of available machines in a NOW will slow parallel jobs because of queuing delay associated with waiting to be scheduled on idle machines. Our study identifies a number of results addressing these problems; they are summarized in Table 1. Essentially, if a number of scheduling pitfalls are avoided, parallel programmers and interactive users can peacefully coexist on a single LAN.

This paper is organized as follows. Section 2 describes our assumptions, methodology, and traces. Our main results are presented in section 3. In section 4, we take our results into account and demonstrate the feasibility of combining both parallel and interactive workloads. Section 5 puts the study in the context of related work. The paper concludes with a summary of our results.

2 Methodology

In this section, we give an overview of the methodology used in this study. In capsule, we traced a research cluster of workstations running interactive jobs and a production MPP running parallel jobs, and then simulated the behavior of combining the two workloads.

Our trace of workstation activity records the following values every two seconds: average system and user cpu utilization, amount of active and mapped physical memory, local disk activity, and terminal activity. This information is written to disk utilizing less than 1% of local disk bandwidth. The trace program itself occupies 500 KB of physical memory and uses two to three percent of the CPU over the two second period. We also maintain a log of all keyboard and mouse activity. Every time a user starts typing or moves the mouse, we log an "active" event. If the keyboard and mouse are both idle for 60 seconds, an "inactive" event is recorded.

The traced cluster consisted of a network of 53 DECstation 5000s. Each of these workstations contains a MIPS R3000 processor, 64 MB of memory, a local disk, and runs the Ultrix 4.3 operating system. The cluster is connected by a 100 Mbps FDDI ring. The users of these workstations are electrical engineers who are members of the CAD group at UC Berkeley. This cluster is considered to be one of the most heavily utilized at Berkeley. Data was collected for over one month, during February and March of 1994. The workstation traces used in our simulations are composed from two weekdays separated by two weeks, allowing the use of 106 workstations in our simulations; unless otherwise noted, the simulator uses traces from 60 workstation-days.

To characterize the resource demands of parallel programs on an MPP, we traced the job submission logs from the Los Alamos National Laboratories' CM-5 for the month of October, 1993. Each job submitted to the CM-5 has an entry in the log of the following form: submission time, CPU time required, number of nodes utilized, and average memory used per node. Values for disk and network activity on the CM-5 were not available. This particular CM-5 has 1024 nodes, in static partitions of size 512, 256, 128, 64, and two of size 32. Unless otherwise noted, the

simulator uses the job submission patterns of one of the two 32 node partitions for a particularly busy weekday (October 26, 1993). Using different days at LANL or different workstation traces did not produce qualitatively different simulation results.

The results presented in this paper are obtained from a simulator that schedules the LANL parallel jobs with the sequential workload. Parallel jobs are run one at a time in a round-robin fashion with a two second time quanta. The simulator produces two metrics for evaluating the performance of a NOW: normalized parallel program slowdown and the number of times a user is delayed by parallel programs. Non-normalized slowdown is defined as the simulated "wall clock" run time of a job divided by the CPU time. Slowdown can be caused by multiprogramming among parallel jobs as well as the effects of the non-dedicated NOW environment. To isolate the latter, slowdown on a NOW is normalized by dividing by the slowdown on a dedicated MPP. For example, if a job experiences a slowdown of 2 due to queuing delay on an MPP and a slowdown of 3 on a NOW, we report a normalized 50% slowdown.

Delays to interactive users are the other main effect of running parallel programs on a NOW. For example, on returning to their workstations, users may notice a significant delay because their machine's virtual memory and file cache were flushed by the execution of a parallel program. Therefore, our simulation tracks the number of such potential user delays to quantitatively determine the effect of different scheduling policies.

The simulator assumes that parallel performance on a NOW is similar to that of an equal-sized MPP. This is optimistic in assuming that the communication performance of parallel jobs is identical in both environments and pessimistic in assuming that CPU performance is identical on both platforms[1]. Also, we assume that network contention between parallel and sequential jobs does not degrade communication performance. The simulator also assumes that when one of the workstations used by a parallel job becomes unavailable (e.g. an interactive user resumes work), the parallel program is halted until another available workstation is found. The process state is then saved and migrated [Theimer et al. 1985, Douglis & Ousterhout 1991] to the new node.

3 Results

In this section, we present our results on how resources should be scheduled on a NOW. The first of these indicates when a system needs to harvest cycles for parallel programs, the next two show how combining both parallel and sequential loads onto the same system may impact both parallel and interactive users, and the last two suggest techniques for scheduling resources in a NOW.

3.1 Response time is important for both parallel and sequential programmers

It has long been understood that the majority of people use their workstations during the day-time hours. Figure 1 shows that this also holds true for the group of workstations we study. Over a one week period, only 5% of keyboard and mouse activity occured between 1 AM and 8 AM.

Given such light night-time machine usage, one approach to combining parallel and sequential workloads on a single platform is to run the parallel jobs only at night. With this simple approach, interactive users will not feel the effect of parallel jobs. Further, parallel programmers can run their code without worry of being disrupted by sequential jobs, allowing them access to a "supercomputer by night".

[1] Typically, MPP processors lag behind workstation processors in performance because of MPP system-development time.

Result	Consequence if ignored
Parallel programmers want more than a "supercomputer by night"	Parallel programmers dissatisfied: they work during day as well
Delays to interactive users must be quantified and avoided	Users are interrupted frequently: will not want machine donated to pool
Efficient context swap of memory necessary for performance	Parallel and sequential performance suffers
Definition of an available machine should be empirically derived	Parallel and sequential performance suffers
A social contract can be used to limit user delays	Some users delayed much more often than others

Table 1: *Five results and their consequences if they are ignored in scheduling resources in a network of workstations.*

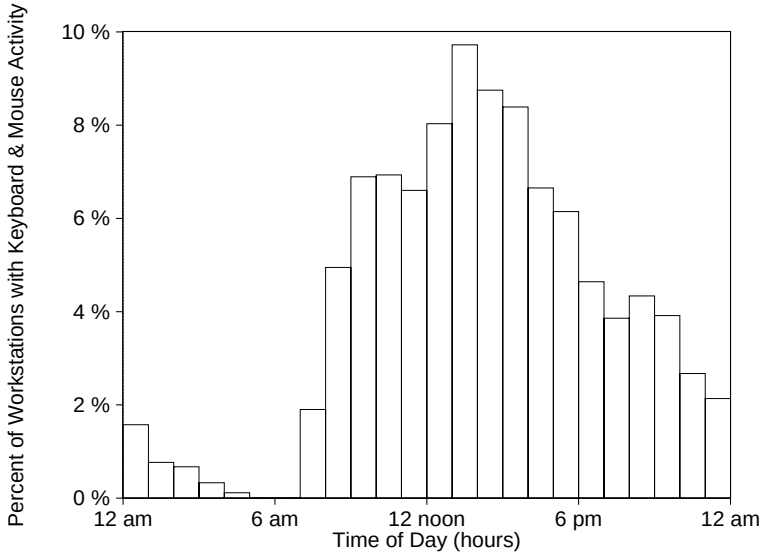

Figure 1: *Keyboard and Mouse Activity.* The plot shows the percent of all keyboard and mouse activity that occurs during each hour of the day. For example, 8% of activity occured between 12 noon and 1 PM. The sum of the bars equals 100 percent. Data is derived from one week of traces on the Berkeley CAD group cluster.

Unfortunately, this solution leaves the parallel-programming community unhappy (and perhaps sleep-deprived). Figure 2 plots the submission times of jobs executed on all partitions of Los Alamos National Laboratories' CM-5 over a one month period. Not surprisingly, parallel programmers also work during the day, with only 23% of MPP jobs submitted between 1 AM and 8 AM. This pattern occurs despite active economic encouragement by LANL management to submit longer running parallel jobs at night. This result is further evidence that parallel programmers also work in the edit/compile/debug cycle; 90% of all the CM-5 jobs complete in less than two minutes. Quick response time is thus a boon for both sequential and parallel programmer productivity.

3.2 A parallel workload can significantly impact interactive users

With the previous result in mind, NOWs may be built to maximize the amount of resources recruited for parallel programs. Undoubtedly, parallel program throughput is a key metric for evaluating the success of a NOW. However, an aggressive machine recruitment policy can lead to frequent delays to interactive users.

Here, *user delays* are defined as times when interactive users return to their workstation to find a parallel job running on their machines or their workstation state changed by a parallel job that ran to completion. These delays consist of two components: the time to import the user's previous context and the time to export the parallel job from the machine. A machine *recruitment policy* is the technique used for identifying a workstation as available for running a parallel job. For example, one policy might consider a machine available if its average CPU utilization is less than 20% for at least one minute.

Figure 3 shows a cumulative graph on how two recruitment policies affect the number of delays to interactive users. The format of all graphs showing user delays in this paper are identical: the x-axis shows the number of daily user delays while the y-axis shows the percentage of users who suffer at least a given number of delays. The first recruitment policy, "CPU only", classifies a machine as available if its average CPU utilization is less than 20% for a one minute period. With such a policy, one unlucky user would be delayed 80 times during the day on our measured workload. The second recruitment policy uses a method similar to that in Condor [Mutka & Livny 1991]: a machine is available if it's average CPU utilization is less than 20% and there is no keyboard activity, both for a minimum of 15 minutes. This user-sensitive policy does significantly better; however, there remain three users who are delayed at least 15 times during the day. In the next subsection, we show that each of these delays can be potentially quite lengthy.

3.3 Fast context swap is important for both parallel program throughput and interactive response times

Two components contribute to the length of interactive user delays. First, the parallel process must be migrated from the workstation and second, the interactive user's previous CPU, memory, and file cache contexts must be restored. These costs may have adverse affects upon not only interactive users but to the parallel program performance as well.

First, we examine the affects of migration. One hypothesis states that since migration is the rare case [Eager et al. 1986, Douglis & Ousterhout 1991], its performance need not be optimized. To test this hypothesis, we varied the cost of process migration in our simulator. Figure 4 shows that parallel programs on systems with high migration costs suffer an average slowdown of up to 34% more than systems with lower migration costs. This slowdown can be attributed to the larger "dead time" the parallel program experiences while the program waits for the migration to

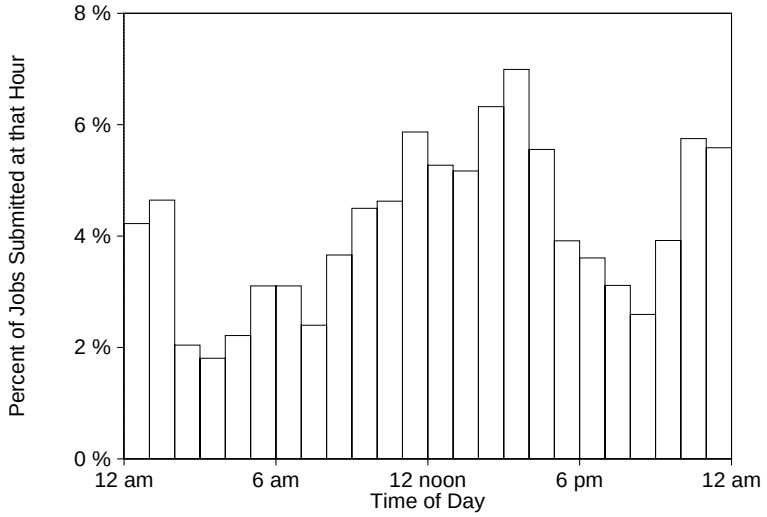

Figure 2: *Plot of submission times at LANL.* Each bar shows the percent of all jobs submitted in a particular hour, over a one month period. Data was taken from all partitions.

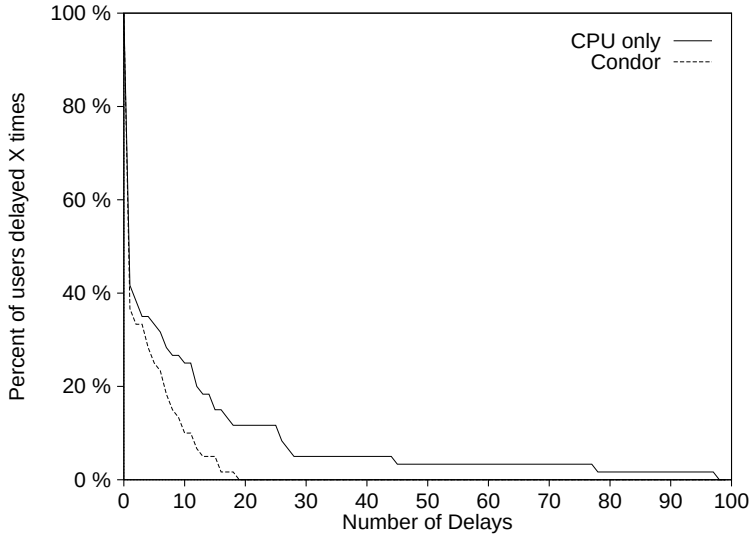

Figure 3: *Plot of Delays per Day.* The x-axis shows the number of times a user gets delayed, and the y-axis shows the percent of times a user gets delayed x times. The CPU only line uses a definition that runs jobs on machines when there is no load, while the 'Condor' line uses a definition that is sensitive to both CPU and keyboard input.

complete before continuing. For example, Litzkow and Solomon report that the Condor system requires two minutes for migration [Litzkow & Solomon 1992], which corresponds to a 28% parallel program slowdown when compared to a system where process migration takes two seconds. High migration costs can also have detrimental effects on interactive users as they may have to wait longer for parallel processes to be evicted from their workstations before resuming their work.

The cost of restoring a user's working set consists mainly of restoring the physical memory. Consider the four main resources of the workstation: CPU, main memory, disk, and network interface. Reclaiming the CPU is inexpensive; restoring the state of registers, cache and TLB costs no more than a few milliseconds. A similar argument can be made for the network interface because little state changes upon a context switch. However, a considerably more state is associated with the workstation's main memory and file cache.

Figure 5 shows that replacing main memory pages can be quite costly. Interactive users will quickly become frustrated if they expect near-instantaneous response times and instead must wait for their state to be swapped back each time they resume work on their workstation.

Table 2 quantifies the effect of flushing a machine's main memory and/or file cache. Flushing either one can have significant effect on the execution time of some typical UNIX programs. Depending on the application, slowdowns from 1 to 45 percent were observed. In an attempt to simulate user activity, the last benchmark, *user*, is a shell script composed of typical Unix commands (`ls`, `grep`, `make`, etc.) based on the commands found in the SPEC SDET benchmark.

Unlike the process migration case, there is no obvious solution to this problem. Including a faster network paging scheme could help, as could pinning "important" memory pages down (e.g. the X server, emacs code pages, etc.). Further, avoiding recruitment of machines with large amounts of active memory should result in less costly delays to the user.

3.4 It is possible to empirically derive the definition of an available machine

All systems that attempt to harvest available machines in a NOW must determine a working definition for machine availability. Unfortunately, much of the previous work in this area relies upon either synthetic workloads or *ad hoc* experimental methods to derive the definition of availability as we show in Section 5. We avoid this pitfall by providing a methodology for determining whether a machine is idle: identify a typical cluster workload and determine the profile of the target workload. For our target cluster, we determined a typical workload to be an application suite that includes: compilation of the program `trn`, LaTeX of a 300 page document, and typing at an `emacs` window. The graphs in Figure 6 demonstrate the CPU and memory profile for running this workload.

From the profile, we can empirically define an instantaneous definition of machine availability for our workload: the methodology requires the definition of idle to be derived for each distinct workload. On our particular cluster, an idle workstation must have CPU utilization less than 20%, constant memory usage less than 20% of available memory, and no keyboard or mouse activity. Since CPU utilization for emacs hovers in the 10-20% range, it might appear that using a CPU threshold of 10% in the definition of an available machine is more appropriate. Unfortunately, regular system daemon activity also produces CPU spikes of 10-20%. While it would be possible to account for these spikes, it is simpler to set the CPU threshold to 20% and allow any keyboard and mouse activity to mark a machine as unavailable.

Given this instantaneous definition of an available machine, we can also quantitatively derive the *recruitment threshold*, or the amount of time the system should wait before attempting to actually harvest the machine. Intuitively, choosing the a small recruitment threshold would maximize

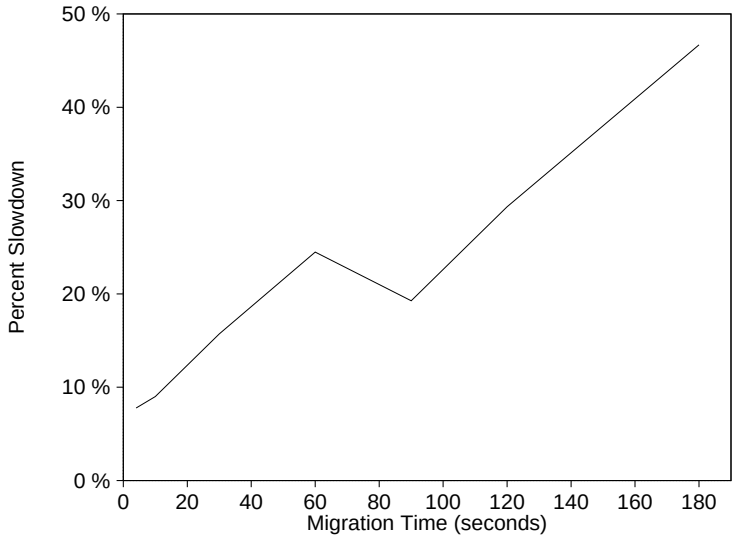

Figure 4: *Effects of the cost of process migration.* The figure demonstrates parallel program slowdown as a function of the cost of process migration.

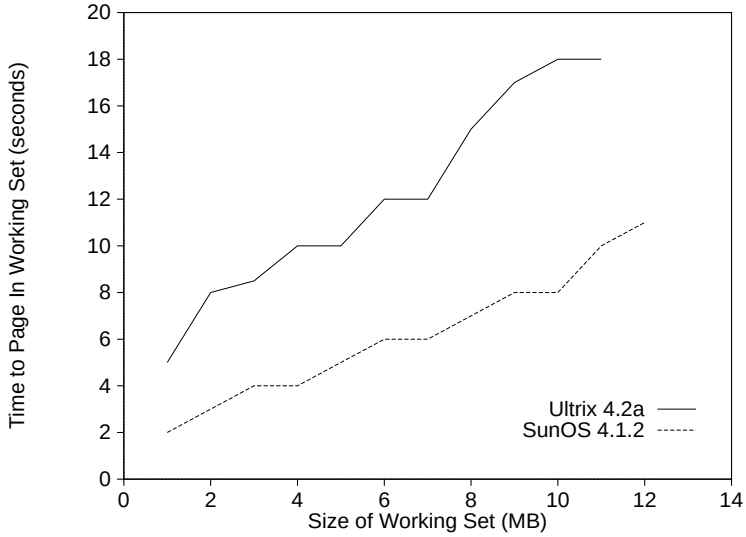

Figure 5: *The Cost of Paging.* A program which utilized a varying amount of memory was run, then stopped. A cleaner that uses all of physical memory was then run. After it completed, the original program was continued, and the time to page in the working set was measured by monitoring local disk activity. Experiments were performed on a Sun IPX running SunOS 4.1.2 and a DECstation 5000/200 running Ultrix 4.2a.

Program	Run Time (seconds)	Slowdown (file cache cleaned)	Slowdown (memory cleaned)	Slowdown (both cleaned)
LaTeX	15.0	18.2%	27.3%	45.5%
make	42.0	31.0%	30.9%	39.1%
gzip	23.3	1.0%	8.7%	8.7%
user	101.7	8.9%	2.0%	12.9%

Table 2: *Replacement Effects.* This table displays the effects of cleaning the file cache, memory, or both on the run time of each of the benchmarks. Experiments were performed on a DECstation 5000/200 running Ultrix 4.2a.

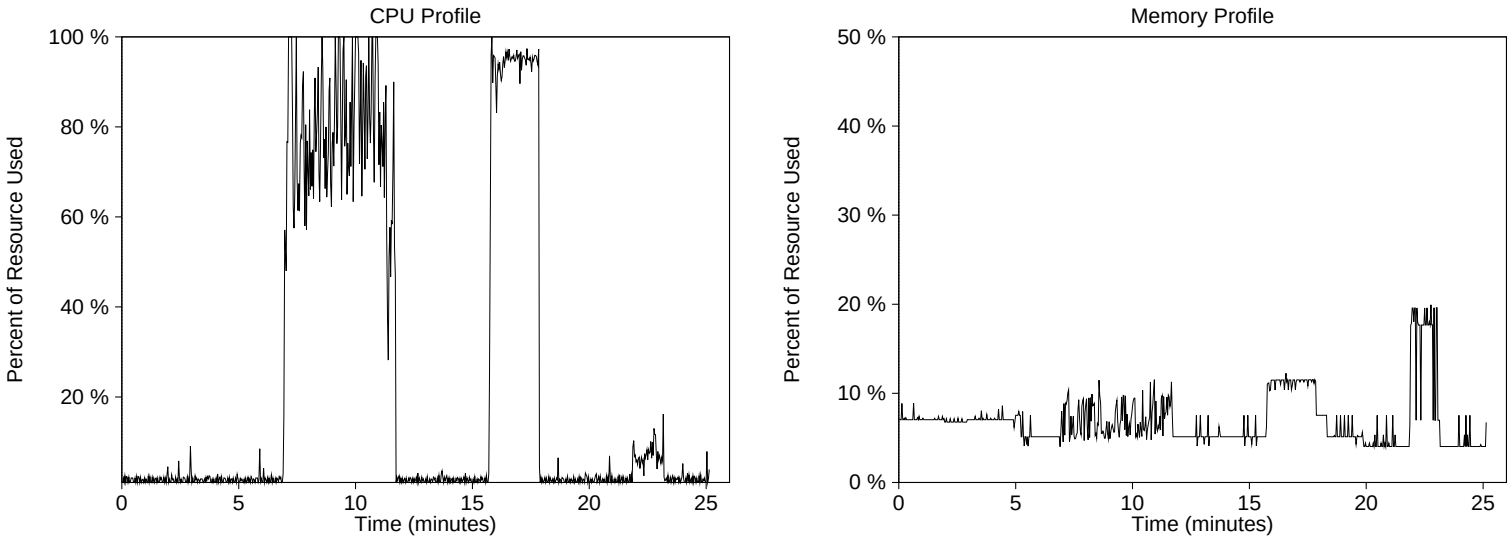

Figure 6: *Profiling a workload.* Shows the behavior of programs as the trace program sees them. On the left is CPU behavior during runs of gcc (from the 7 to the 12 minute mark), LaTeX(16 to 18), and typing in emacs (23 to 24). On the right is the corresponding memory behavior. Experiments performed on a DECstation 5000/200 running Ultrix 4.2a.

parallel program throughput. Douglis and Ousterhout [Douglis & Ousterhout 1991] demonstrate that machines that have only recently gone idle should be avoided since they are most likely to once again become unavailable. Our simulations confirm this result; Figure 7 plots parallel program slowdown as a function of recruitment threshold. A 180 second recruitment threshold maximizes parallel program throughput. This value ensures that the machine is likely to remain available, and yet does not lead the system to squander a large amount of time that could have been used to run parallel programs.

3.5 A social contract: limiting user delays with minimal effect on parallel performance

In this subsection, we demonstrate that a very simple scheduling mechanism can vastly reduce the worst case number of user delays without adversely affecting the performance of parallel programs. A *social contract* system guarantees that an individual will not be delayed more than a specified number of times during any day. Once a user has been delayed the threshold number of times, that user's workstation will no longer be a candidate to run parallel programs.

Figure 8 shows the percentage of parallel program that complete with the 24 hour period as a function of the social contract value and the number of user delays. For social contract values greater than 5, the normalized parallel program slowdown presented is uniform at approximately 10% and almost all submitted parallel jobs are completed. Once the maximum number of user delays drops below 5 however, more than half of the jobs in our parallel workload do not complete. Thus, the social contract identifies a tradeoff between maximum daily user delays and parallel program throughput. Further, it distributes delays more uniformly among workstations.

4 Putting it all together

The previous section demonstrates that the definition of an available machine can be experimentally derived and that through a social contract the number of user delays can be limited without affecting parallel program throughput. Building upon these results, we set out to determine how many machines are needed to support the 32-node LANL workload as well as the sequential jobs of interactive users. Intuitively, as the number of machines available to run the parallel workload decreases, the number of user delays will increase.

Figure 9 presents the results of this experiment. The graph shows the number of parallel programs that did not complete as a function of the social contract. If available machines are abundant, then the slowdown is independent of the social contract value, and a large fraction of jobs complete. For that range, we find that slowdown is consistently about 10%. As machines become more scarce, performance becomes highly sensitive to the definition of the social contract. The graph shows that if only 10 daily user delays are allowed, at least 54 machines must be available to run the workload; for less than 54 machines, a significant number of jobs do not finish in the allotted time. If up to 20 daily user delays are tolerated, then 32-node parallel jobs can be run on 50 workstations without noticeable slowdown.

5 Related and Future Work

There have been an abundance of studies on workstation clusters and their potential to support distributed and parallel computing. Theimer and his colleagues estimated that roughly one-third of their 25 workstations were free, even at the busiest times of the day [Theimer & Lantz 1989].

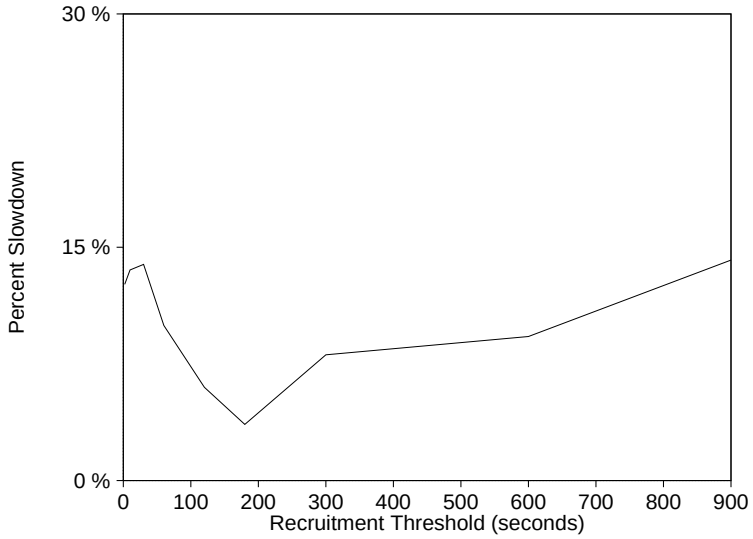

Figure 7: *Recruitment threshold results.* The figure shows parallel program slowdown as a function of the recruitment threshold. The minima is the point where a machine is likely to remain idle for a long period of time.

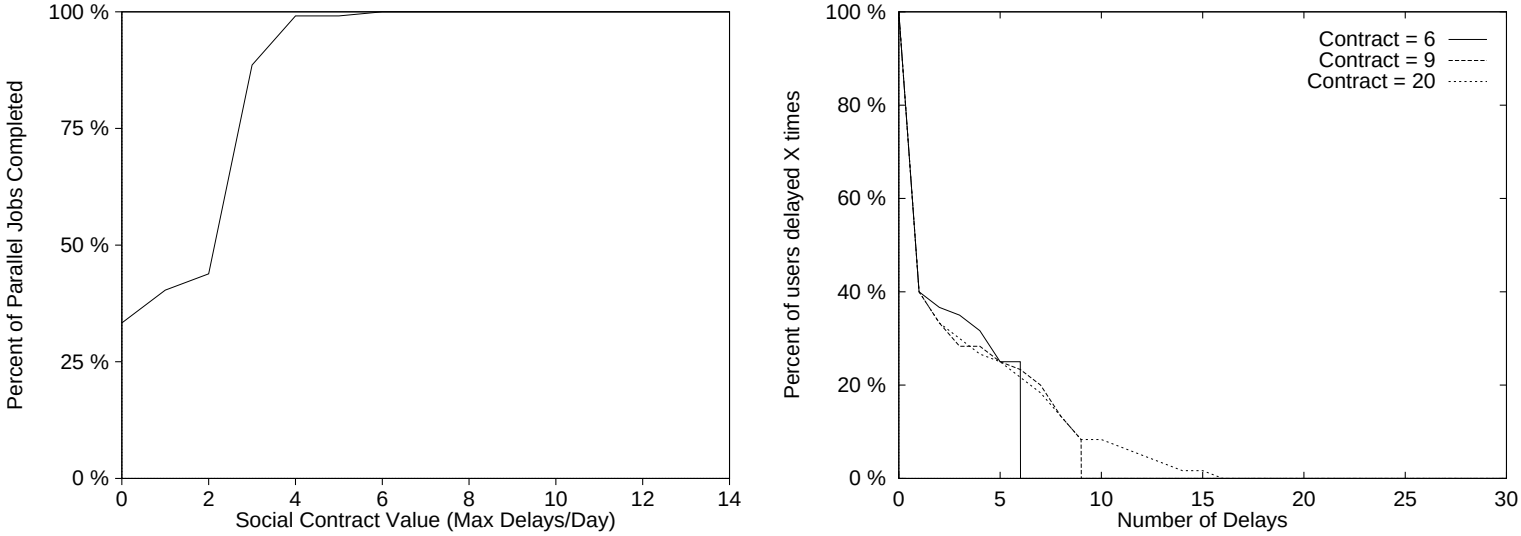

Figure 8: *Social Contract Results.* The left-hand graph shows the effect the social contract has upon the percentage of parallel jobs that complete in 24 hours. On the right is a cumulative plot of the percentage of users which were delayed at least a give number of times during a 24 hour period.

Nichols measured 15 to 20% of workstations available during the day, which increased to roughly 30% at night [Nichols 1987]. Douglis and Ousterhout found that about two-thirds of machines were available on average [Douglis & Ousterhout 1991], and Mutka and Livny [Mutka & Livny 1991] found similar results. Table 3 summarizes these results.

We have identified a number of issues that have not been addressed by these previous efforts. First, the only workstation resource monitored is the CPU. While the processor is an important aspect of the system, one should not overlook memory and disk. As far as interactive users are concerned, these two resources may be expensive to reclaim, and thus disrupt the normal work environment.

Second, the definition of an available machine appears somewhat arbitrary, explaining the discrepancies between previous studies. For example, Nichols defines it as having no one logged onto a machine, whereas in Sprite a machine becomes idle after one minute of no keyboard or mouse activity plus a load average of less than 0.3. This difference in definition accounts for the dramatic dissimilarities seen in Table 3. While both Nichols' and Douglis' studies took place in the same year, there is a 40% net difference in their results. We avoid this problem by defining *available* empirically. While it is necessary to understand when available workstations can be utilized, it is not sufficient. Equally important is understanding the impact on interactive users. To address this problem, we evaluate scheduling policies using metrics that include the effects on parallel programs as well as on interactive users, and then use a definition of availability that best suits both communities.

Third, previous studies concentrate on aggregate system availability. We show that it is important to account for more than average workstation idleness. The distribution of idle times—that afternoon idleness is more valuable than night-time idleness and that idle times come in short bursts making the cost of process migration important—is also relevant.

In a related study, Leutenegger et al. [Leutenegger & Sun 1993] use simulation to study whether parallel programs can run in a non-dedicated environment (such as a NOW). While demonstrating that this may be possible without a significant impact on the parallel programs, their work does not discuss any potential impact upon interactive users. Further, all the simulations are driven by synthetic models of both workstation and parallel program behavior. Our study makes use of real traces from both a network of workstations and a parallel machine, and quantifies the number of interruptions experience by interactive users.

In the future, we plan on profiling workstations at several industrial sites: Hewlett Packard Laboratories, Sun Laboratories, and SynOptics corporation. While we have taken traces of a number of other workstation clusters around the Berkeley campus and found that the usage patterns closely match the CAD cluster, we wish to determine if our measurements generalize beyond research/educational environments.

6 Conclusions

By making use of trace-driven simulation, we conclude that a network of workstations in an academic environment can sustain both an interactive and parallel workload. We identified two metrics in evaluating the success of a particular scheduling policy: parallel program throughput and delays suffered by interactive users. With the assistance of these metrics, we discovered five results relevant to NOW system designers.

The first result shows it is necessary to provide a "supercomputer by day" as well as by night, since parallel programmers work during the day time and demand interactive use of the system. The second and third results demonstrate that simple machine recruitment policies will delay some

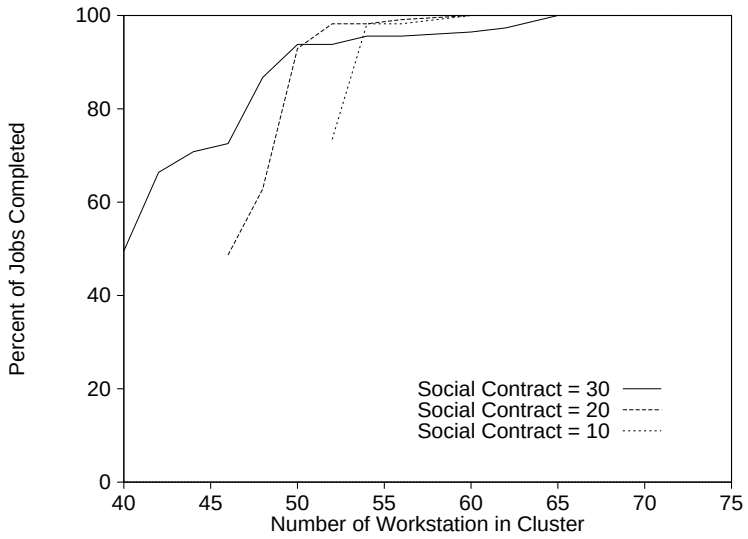

Figure 9: *Workstations Necessary to Run Parallel Workload.* The graph indicates how different social contract values affect parallel program throughput. Note that with lower social contract values, more workstations must be allowed into the cluster to handle the parallel workload.

Who	Year	Idle Definition	Free (day)	Free (night)
Theimer et al.	1984	Not described	33%	N/A
Nichols	1986	No one logged in	15-20%	30%
Douglis et al.	1986	1 min idle,no load	66%	78%
Mutka and Livny	1987	1 min low cpu avg.	60%	80%

Table 3: *Related work.* How previous efforts have defined idle and the consequences of those definitions.

users a noticeable number of times, and that each delay could be costly to both interactive users and parallel programs. Next, we provide a methodology for empirically deriving the definition of an available machine: identifying a typical cluster workload and setting machine availability thresholds (CPU and memory usage, keyboard activity, etc.) below values found in the profile for the given workload. Finally, a simple scheduling policy, the *social contract*, limits the number of delays any one user feels while preserving parallel job throughput.

References

[Biagioni et al. 1993] Biagioni, E., Cooper, E., and Sansom, R. Designing a practical ATM LAN. *IEEE Network*, 7(2), 1993.

[Blumrich et al. 1994] Blumrich, M., Li, K., Alpert, R., Dubnicki, C., Felton, E., and Sandberg, J. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[Carriero & Gelernter 1989] Carriero, N. and Gelernter, D. Linda in Context. *Communications of the ACM*, April 1989.

[Cohen et al. 1993] Cohen, D., Finn, G. G., and Felderman, R. ATOMIC: A Very-High-Speed Local Communication Architecture. In *International Conference on Parallel Processing*, August 1993.

[Douglis & Ousterhout 1991] Douglis, F. and Ousterhout, J. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–85, August 1991.

[Eager et al. 1986] Eager, D. L., Lazowska, E. D., and Zahorjan, J. Adaptive Load Shating in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.

[Feitelson & Rudolph 1992] Feitelson, D. G. and Rudolph, L. Gang Scheduling Performance Benefits for Fine-Grained Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–18, December 1992.

[Gelernter 1985] Gelernter, D. Parallel Programming in Linda. In *Proceeding of the International Conference on Parallel Processing*, pp. 255–263, August 1985.

[Gupta et al. 1991] Gupta, A., Tucker, A., and Urushibara, S. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proceedings of the ACM SIGMETRICS Conference*, pp. 120–32, May 1991.

[Kronenberg et al. 1986] Kronenberg, N. P., Levy, H. M., and Strecker, W. D. VAXclusters: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems*, 4(2), 1986.

[Leutenegger & Sun 1993] Leutenegger, S. T. and Sun, X.-H. Distributed Computing Feasibility in a Non-Dedicated Homogenous Distributed System. In *Supercomputing 93*, 1993.

[Litzkow & Solomon 1992] Litzkow, M. and Solomon, M. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. In *Winter 1992 USENIX Conference*, pp. 283–290, January 1992.

[Mutka & Livny 1991] Mutka, M. M. and Livny, M. The Available Capacity of a Privately Owned Workstation Environment. *Performance Evaluation*, 12(4):269–84, July 1991.

[Nichols 1987] Nichols, D. Using idle workstations in a shared computing environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pp. 5–12, November 1987.

[Ousterhout 1982] Ousterhout, J. K. Scheduling Techniques for Concurrent Systems. In *Third International Conference on Distributed Computing Systems*, pp. 22–30, May 1982.

[Sunderam 1990] Sunderam, V. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[Theimer & Lantz 1989] Theimer, M. M. and Lantz, K. A. Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Transactions on Software Engineering*, 15(11):1444–57, November 1989.

[Theimer et al. 1985] Theimer, M., Landtz, K., and Cheriton, D. Preemptable Remote Execution Facilities for the V System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pp. 2–12, December 1985.